Speed up ASP.NET Core WEB API application. Part 1.

Eduard Silantiev, 23 Sep 2018
https://www.linkedin.com/in/eduard-silantiev-012922148/

Source code on GitHub

Create fast RESTful WEB API service using ASP.NET Core 2.1

Introduction

In this article we review the process of creating ASP.NET WEB API application using ASP.NET Core. The main focus will be on the application productivity.

The article is divided into two parts:

- Part 1. Creating a test RESTful WEB API application;
- Part 2. Using various approaches to increase the application's productivity.

Part 1. Creating a test RESTful WEB API service.

In Part 1 we will create an asynchronous RESTful WEB API service that be able to search products in a database and get price lists of different suppliers for the particular product.

For coding we need Microsoft Visual Studio 2017 (updated to .NET Core 2.1) and Microsoft SQL Server (any version)

We will review the following:

The application architecture

We will build our WEB API application using Controllers – Services – Repositories – Database architecture.

Controllers are responsible for routing – they accept http requests and invoke appropriate methods of services with parameters received from the request parameters or body. By convention, we named classes that encapsulate business logic as “Services”. After processing a request, a service returns a result of IActionResult type to a controller. The controller does not care about the type of service result and just transmits it to the user with the http response. All methods of receiving the data from or storing the data in a database are encapsulated in Repositories. If the Service needs some data, it requests the Repository without knowing where and how the data is stored.

This pattern provides maximum decoupling of application layers and makes it easy to develop and test the application.

The Database

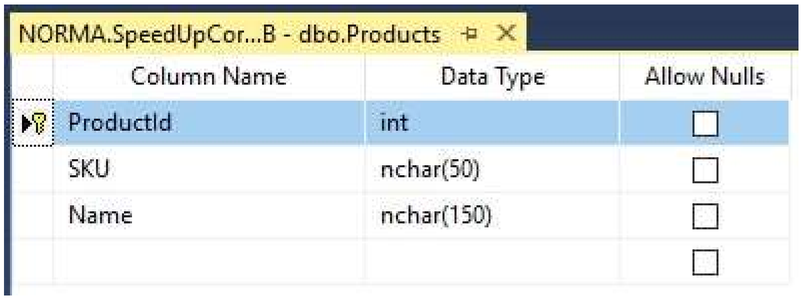
In our application we use a Microsoft SQL Server. Let us create a database for our application and fill it with test data and execute the next query in the Microsoft SQL Server Management Studio:

Hide Shrink Copy Code

```
USE [master]
GO

CREATE DATABASE [SpeedUpCoreAPIExampleDB]
GO

USE [SpeedUpCoreAPIExampleDB]
GO

CREATE TABLE [dbo].[Products] (
    [ProductId] INT         IDENTITY (1, 1) NOT NULL,
    [SKU]       NCHAR (50)  NOT NULL,
    [Name]      NCHAR (150) NOT NULL,
    CONSTRAINT [PK_Products] PRIMARY KEY CLUSTERED ([ProductId] ASC)
);
GO

CREATE TABLE [dbo].[Prices] (
    [PriceId]   INT             IDENTITY (1, 1) NOT NULL,
    [ProductId] INT             NOT NULL,
    [Value]     DECIMAL (18, 2) NOT NULL,
    [Supplier]  NCHAR (50)      NOT NULL,
    CONSTRAINT [PK_Prices] PRIMARY KEY CLUSTERED ([PriceId] ASC)
);
GO

ALTER TABLE [dbo].[Prices]  WITH CHECK ADD  CONSTRAINT [FK_Prices_Products] FOREIGN
KEY([ProductId])
REFERENCES [dbo].[Products] ([ProductId])
ON DELETE CASCADE
GO
ALTER TABLE [dbo].[Prices] CHECK CONSTRAINT [FK_Prices_Products]
GO

INSERT INTO Products ([SKU], [Name]) VALUES ('aaa', 'Product1');
INSERT INTO Products ([SKU], [Name]) VALUES ('aab', 'Product2');
INSERT INTO Products ([SKU], [Name]) VALUES ('abc', 'Product3');

INSERT INTO Prices ([ProductId], [Value], [Supplier]) VALUES (1, 100, 'Bosch');
INSERT INTO Prices ([ProductId], [Value], [Supplier]) VALUES (1, 125, 'LG');
INSERT INTO Prices ([ProductId], [Value], [Supplier]) VALUES (1, 130, 'Garmin');

INSERT INTO Prices ([ProductId], [Value], [Supplier]) VALUES (2, 140, 'Bosch');
INSERT INTO Prices ([ProductId], [Value], [Supplier]) VALUES (2, 145, 'LG');
INSERT INTO Prices ([ProductId], [Value], [Supplier]) VALUES (2, 150, 'Garmin');

INSERT INTO Prices ([ProductId], [Value], [Supplier]) VALUES (3, 160, 'Bosch');
INSERT INTO Prices ([ProductId], [Value], [Supplier]) VALUES (3, 165, 'LG');
INSERT INTO Prices ([ProductId], [Value], [Supplier]) VALUES (3, 170, 'Garmin');

GO
```

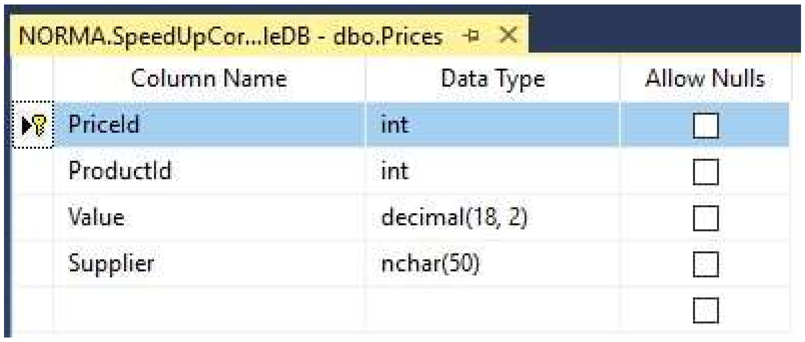
Now we have a database with the name SpeedUpCoreAPIExampleDB, filled with the test data

The Products table consists of a products list. The SKU field is for searching for products in the list.


NORMA.SpeedUpCor...B - dbo.Products

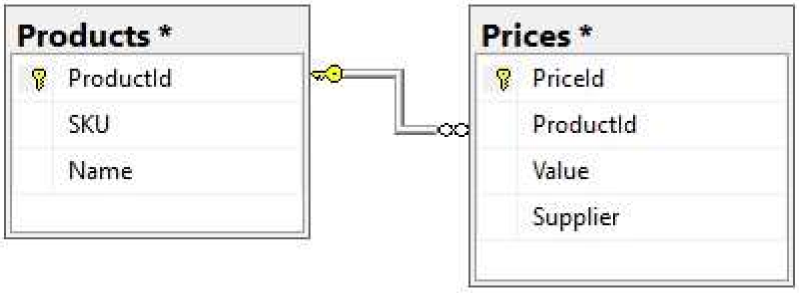
Column Name	Data Type	Allow Nulls
ProductId	int	☐
SKU	nchar(50)	☐
Name	nchar(150)	☐
		☐

The Prices table consists of a list of prices.


NORMA.SpeedUpCor...leDB - dbo.Prices

Column Name	Data Type	Allow Nulls
PriceId	int	☐
ProductId	int	☐
Value	decimal(18, 2)	☐
Supplier	nchar(50)	☐
		☐

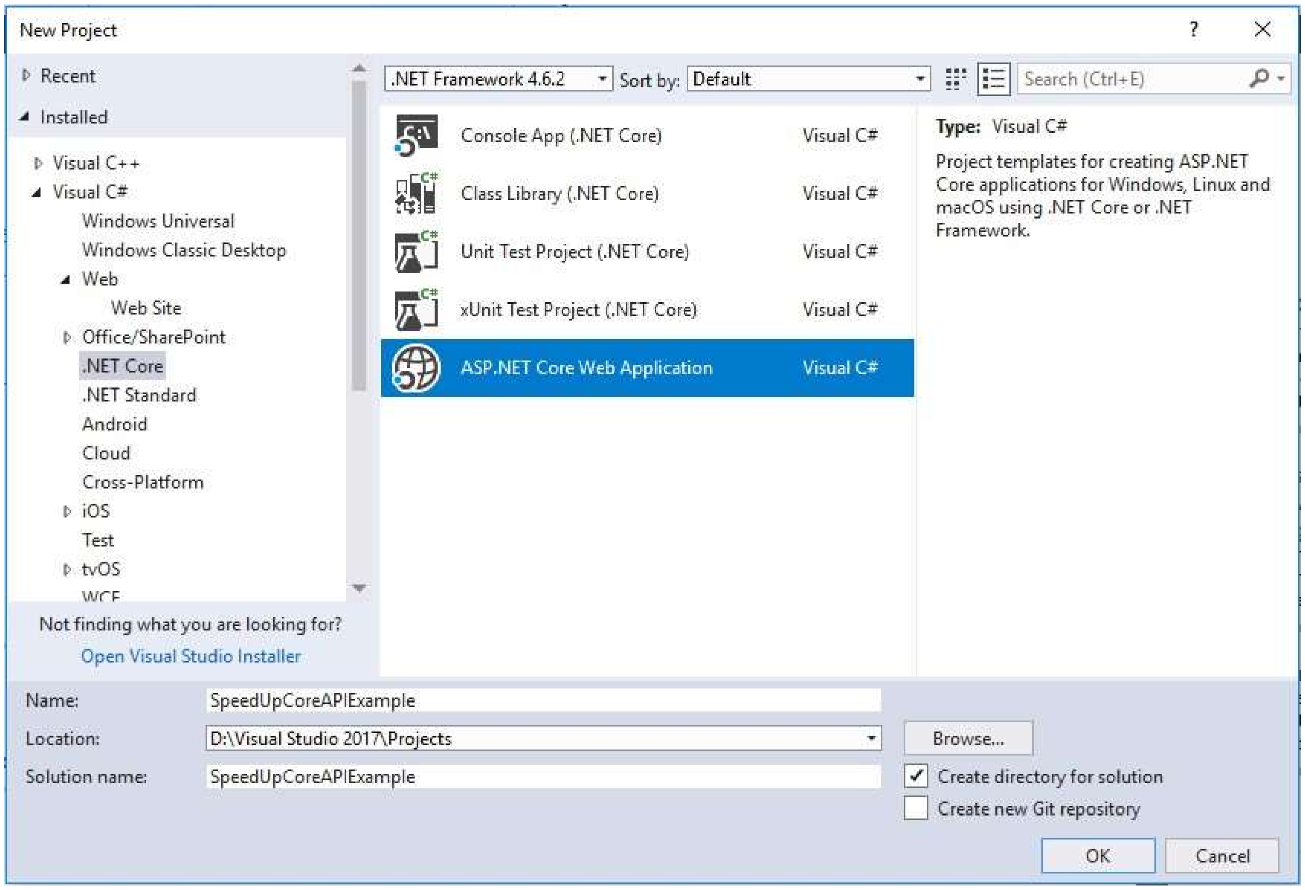
The relationship between these tables can be represented schematically:



Note! We have created a FOREIGN KEY FK_Prices_Products with the CASCADE delete rule so that the MS SQL server be able to provide data integrity between the Products and Prices tables on deleting records from the Products table.

Creating ASP.NET Core WEB API application

In Microsoft Visual Studio start new .NET Core project SpeedUpCoreAPIExample.


New Project

.NET Framework 4.6.2 Sort by: Default

Console App (.NET Core)	Visual C#
Class Library (.NET Core)	Visual C#
Unit Test Project (.NET Core)	Visual C#
xUnit Test Project (.NET Core)	Visual C#
ASP.NET Core Web Application	Visual C#

Type: Visual C#

Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework.

Name: SpeedUpCoreAPIExample

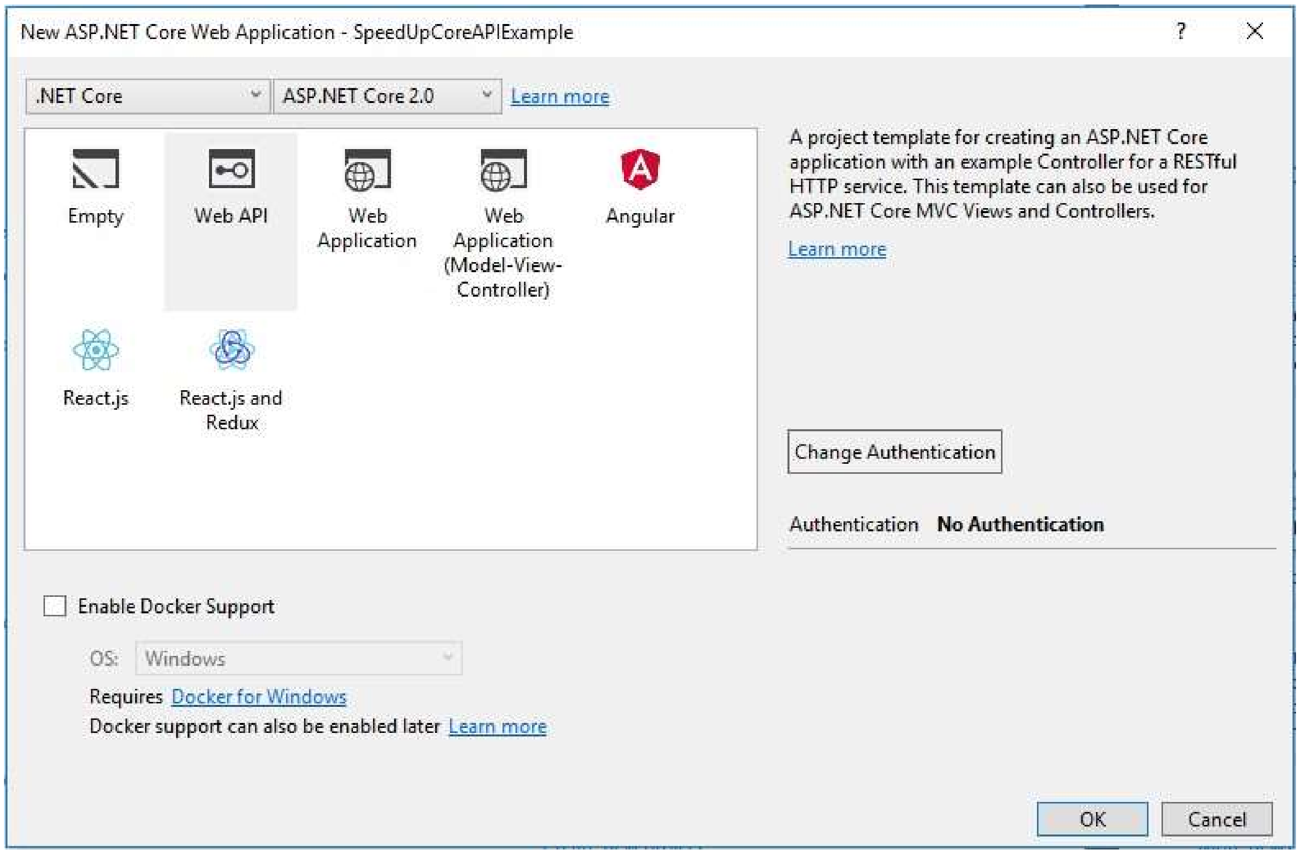
Location: D:\Visual Studio 2017\Projects

Solution name: SpeedUpCoreAPIExample

☑ Create directory for solution

☐ Create new Git repository

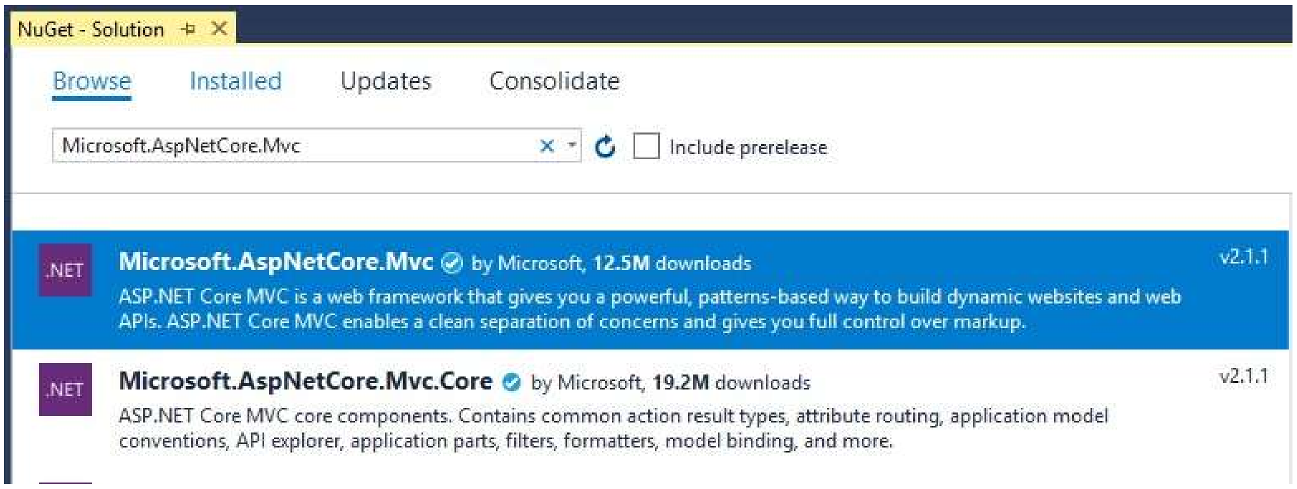
Then select Web API


New ASP.NET Core Web Application - SpeedUpCoreAPIExample
.NET Core
ASP.NET Core 2.0
Learn more
Empty
Web API
Web Application
Web Application (Model-View-Controller)
Angular
React.js
React.js and Redux
Change Authentication
Enable Docker Support
OS: Windows
Requires Docker for Windows
Docker support can also be enabled later Learn more
OK
Cancel

A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Authentication No Authentication

Since we are making a Web API Core application we should install Microsoft.AspNetCore.Mvc NuGet package. Go to menu *Main menu > Tools > NuGet Package Manager > Manager NuGet Packages For Solution* and input Microsoft.AspNetCore.Mvc. Select and install the package:



Microsoft.AspNetCore.Mvc

Microsoft.AspNetCore.Mvc by Microsoft, 12.5M downloads v2.1.1
ASP.NET Core MVC is a web framework that gives you a powerful, patterns-based way to build dynamic websites and web APIs. ASP.NET Core MVC enables a clean separation of concerns and gives you full control over markup.

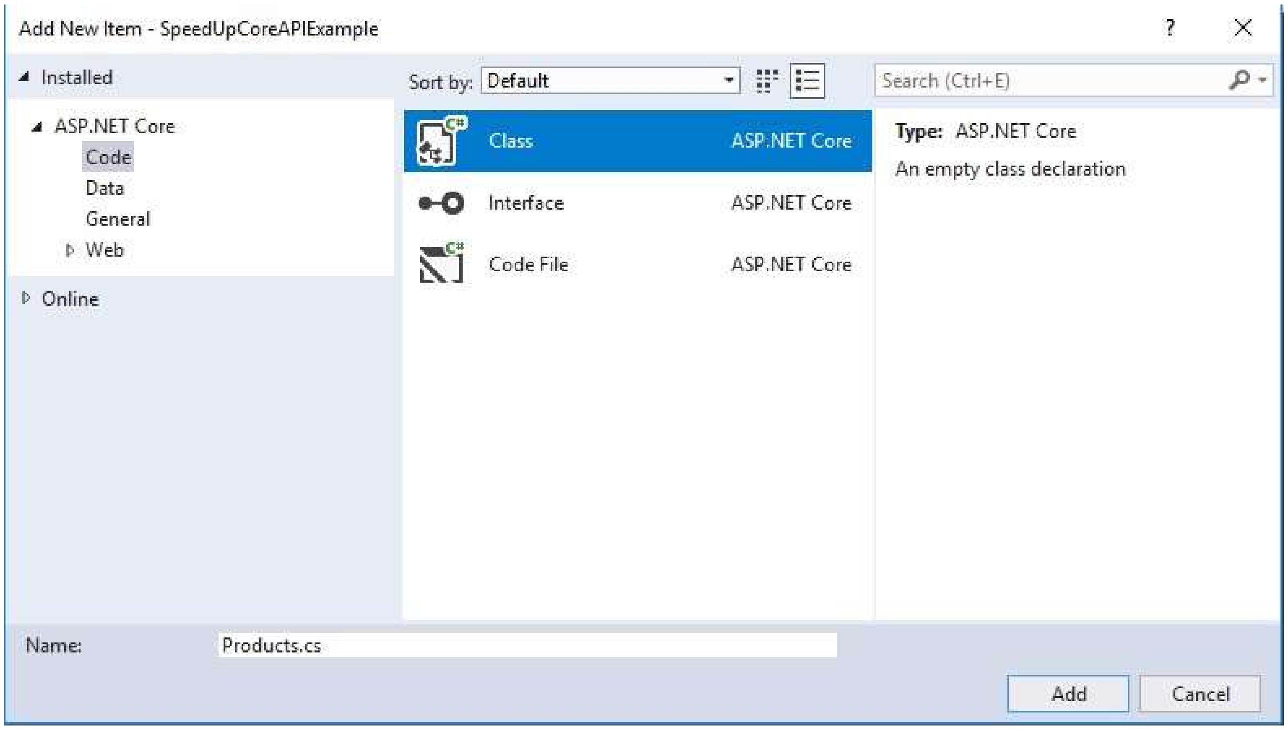
Microsoft.AspNetCore.Mvc.Core by Microsoft, 19.2M downloads v2.1.1
ASP.NET Core MVC core components. Contains common action result types, attribute routing, application model conventions, API explorer, application parts, filters, formatters, model binding, and more.

The next step is to create a data model of our application. Since we have already created the database it seems logical to use a scaffolding mechanism to generate the data model from the database structure. But we will not use scaffolding because the database structure will not entirely reflect the application data model - in the database we follow the convention of naming tables with plural names like "Products" and "Prices", considering the tables to be sets of rows "Product" and "Price" respectively. In our application we want to name entity classes "Product" and "Price", but after scaffolding we would have entities with the names "Products" and "Prices" created and some other objects that reflect the relationship between entities would also be created automatically.

Therefore, we would have to rewrite the code. That is why, we decided to create the data model manually.

In the Solution Explorer, right click your project and select Add > New Folder.

Name it Models. In the Models folder, let us create two entity classes Product.cs and Price.cs. Right click the Models folder then select Add Item > Class > Product.cs


Add New Item - SpeedUpCoreAPIExample

Installed: ASP.NET Core > Code (selected), Data, General, Web; Online

Sort by: Default

Search (Ctrl+E)

Class — ASP.NET Core (selected)
Interface — ASP.NET Core
Code File — ASP.NET Core

Type: ASP.NET Core
An empty class declaration

Name: Products.cs

Enter the text of Product class:

Hide Copy Code

```
namespace SpeedUpCoreAPIExample.Models
{
    public class Product
    {
        public int ProductId { get; set; }
        public string Sku { get; set; }
        public string Name { get; set; }
    }
}
```

And for the Price.cs:

Hide Copy Code

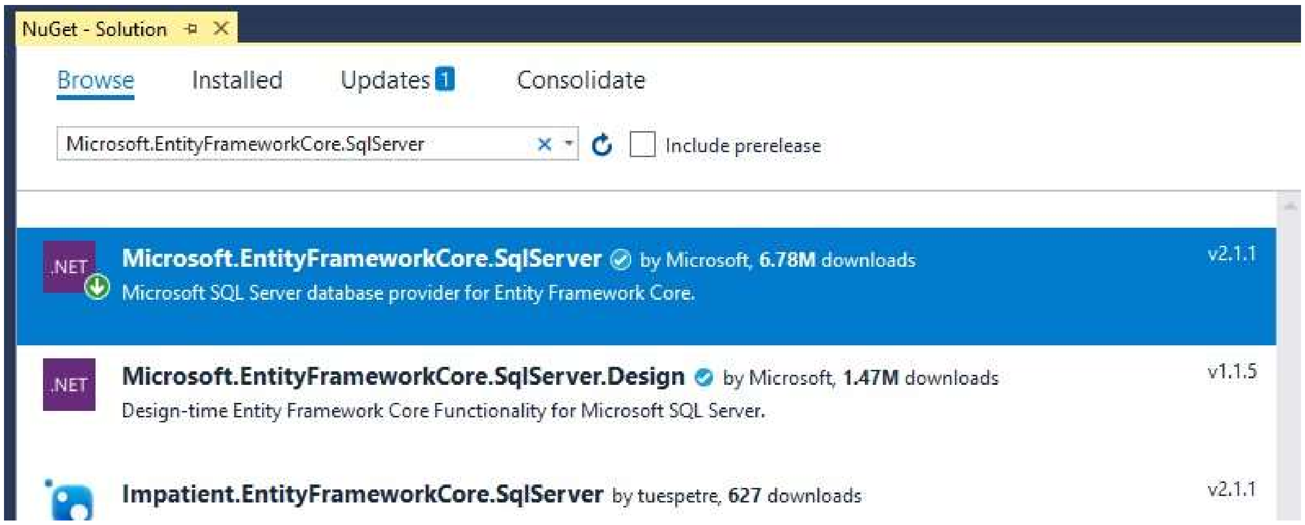
```
namespace SpeedUpCoreAPIExample.Models
{
    public class Price
    {
        public int PriceId { get; set; }
        public int ProductId { get; set; }
        public decimal Value { get; set; }
        public string Supplier { get; set; }
    }
}
```

In the Price class we use the Value field to store price values - we cannot name the field “Price” as fields cannot have the same name as the name of a class (we also use “Value” field in Prices table of our database). Later in the Database context class, we will map these entities to database tables “Products” and “Prices”.

Note that in the model we do not have any relationship between the Products and Prices entities.

Database access with Entity Framework Core

To access the database, we will use Entity Framework Core. For this we need to install an EntityFrameworkCore provider for our database. Go to menu Main menu > Tools > NuGet Package Manager > Manager NuGet Packages For Solution and input Microsoft.EntityFrameworkCore.SqlServer in the Browse field, as we are using a Microsoft SQL Server. Select and install the package:



Microsoft.EntityFrameworkCore.SqlServer by Microsoft, 6.78M downloads v2.1.1
Microsoft SQL Server database provider for Entity Framework Core.

Microsoft.EntityFrameworkCore.SqlServer.Design by Microsoft, 1.47M downloads v1.1.5
Design-time Entity Framework Core Functionality for Microsoft SQL Server.

Impatient.EntityFrameworkCore.SqlServer by tuespetre, 627 downloads v2.1.1

To inform the Entity Framework how to work with our data model, we should create a Database context class. For this, let us create a new folder Contexts, right click on it and select Add > New Item > ASP.NET Core > Code > Class. Name the class DefaultContext.cs

Enter the following text of the class:

Hide Copy Code

```
using Microsoft.EntityFrameworkCore;
using SpeedUpCoreAPIExample.Models;

namespace SpeedUpCoreAPIExample.Contexts
{
    public class DefaultContext : DbContext
    {
        public virtual DbSet<Product> Products { get; set; }
        public virtual DbSet<Price> Prices { get; set; }

        public DefaultContext(DbContextOptions<DefaultContext> options) : base(options)
        {
        }
    }
}
```

In the following lines we mapped data model entities classes to database tables:

Hide   Copy Code

```
public virtual DbSet<Product> Products { get; set; }
public virtual DbSet<Price> Prices { get; set; }
```

According to the Entity Framework Core naming convention for entity key names, model key fields should have name "Id" or EntitynameId (case insensitive) to be mapped by EFC to database keys automatically. We use the "ProductId" and "PriceId" names, which meet the convention. If we use nonstandard names for key fields, we would have to configure the keys in a DbContext explicitly.

The next step for the Database context is to declare it in the Startup class of our application. Open the Startup.cs file in the root of the application and in the ConfigureServices method add "using" directives:

Hide   Copy Code

```
using Microsoft.EntityFrameworkCore;
using SpeedUpCoreAPIExample.Contexts;
```

and correct the ConfigureServices procedure

Hide   Copy Code

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddDbContext<DefaultContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultDatabase")));
}
```

The last step is to configure the Connection string of the database. For this, find the appsettings.json file in the root of our application and add the following ConnectionStrings session:

Hide   Copy Code

```
"ConnectionStrings": {
      "DefaultDatabase": "Server=localhost;Database=SpeedUpCoreAPIExampleDB;Integrated
Security=True;"
}
```

But be aware, that in the root of the application, there is also an appsettings.Development.json configuration file. By default, this file is used during the development process. So, you should duplicate the ConnectionStrings session there or Configuration.GetConnectionString may return null.

Now our application is ready to work with the database.

Asynchronous design pattern

Asynchronously working is the first step of increasing productivity of our application.
Asynchronous design pattern needs some extra coding, but it is worth it - asynchronous methods do not block the application while executing. So, when new requests are received, they are being processed without any delay, therefore without loss of application productivity.
We want all our repositories be working asynchronously, so they will return Task<T> and all methods will have names with the Async suffix. The suffix does not make a method asynchronous. It is used by convention to represent our intentions regarding the method. The combination async – await implements the asynchronous pattern.

Repositories

We will create two Repositories – one for the Product entity and another for the Price entity. We will declare repositories methods in the appropriate interface first, to make the repositories ready for the dependency injection.

Let us create a new folder Interfaces for the interfaces. Right click the Interfaces folder and add a new class with the name IProductsRepository.cs and change its code for:

Hide Copy Code

```
using SpeedUpCoreAPIExample.Models;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Interfaces
{
    public interface IProductsRepository
    {
        Task<IEnumerable<Product>> GetAllProductsAsync();
        Task<Product> GetProductAsync(int productId);
        Task<IEnumerable<Product>> FindProductsAsync(string sku);
        Task<Product> DeleteProductAsync(int productId);
    }
}
```

Then for the IPricesRepository.cs

Hide Copy Code

```
using SpeedUpCoreAPIExample.Models;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Interfaces
{
    public interface IPricesRepository
    {
        Task<IEnumerable<Price>> GetPricesAsync(int productId);
    }
}
```

Repositories implementation

Create a new folder Repositories and add the ProductsRepository class with the code:

Hide Shrink ▲ Copy Code

```
using Microsoft.EntityFrameworkCore;
using SpeedUpCoreAPIExample.Contexts;
using SpeedUpCoreAPIExample.Interfaces;
using SpeedUpCoreAPIExample.Models;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Repositories
{
    public class ProductsRepository : IProductsRepository
    {
        private readonly DefaultContext _context;

        public ProductsRepository(DefaultContext context)
        {
            _context = context;
        }

        public async Task<IEnumerable<Product>> GetAllProductsAsync()
        {
            return await _context.Products.ToListAsync();
        }

        public async Task<Product> GetProductAsync(int productId)
        {
            return await _context.Products.Where(p => p.ProductId ==
productId).FirstOrDefaultAsync();
        }

        public async Task<IEnumerable<Product>> FindProductsAsync(string sku)
        {
            return await _context.Products.Where(p => p.Sku.Contains(sku)).ToListAsync();
        }

        public async Task<Product> DeleteProductAsync(int productId)
        {
            Product product = await GetProductAsync(productId);

            if (product != null)
            {
                _context.Products.Remove(product);

                await _context.SaveChangesAsync();
            }

            return product;
        }
    }
}
```

In the ProductsRepository class constructor, we injected DefaultContext using dependency injection.

Then create PricesRepository.cs with the code:

Hide Copy Code

```
using Microsoft.EntityFrameworkCore;
using SpeedUpCoreAPIExample.Contexts;
using SpeedUpCoreAPIExample.Interfaces;
using SpeedUpCoreAPIExample.Models;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Repositories
{
    public class PricesRepository : IPricesRepository
    {
        private readonly DefaultContext _context;

        public PricesRepository(DefaultContext context)
        {
            _context = context;
        }

        public async Task<IEnumerable<Price>> GetPricesAsync(int productId)
        {
            return await _context.Prices.Where(p => p.ProductId == productId).ToListAsync();
        }
    }
}
```

The last step is to declare our repositories in the Startup class. Add "using" directives in the ConfigureServices method of the Startup.cs:

Hide Copy Code

```
using SpeedUpCoreAPIExample.Interfaces;
using SpeedUpCoreAPIExample.Repositories;
```

and after DefaultContext declaration:

Hide Copy Code

```
services.AddScoped<IProductsRepository, ProductsRepository>();
services.AddScoped<IPricesRepository, PricesRepository>();
```

Note! The sequence of declarations matters for the dependency injection – if you want to inject DefaultContext into repositories, the DefaultContext should be declared before repositories. For repositories we use Scoped lifetime model because the system registers Database context with Scoped model automatically. And a repository that uses the context should have the same lifetime model.

Services

Our “Services” classes encapsulate all the business logic except accessing the database – for this they have repositories injected. All services methods run asynchronously and return IActionResult depending on the result of data processing. Services handle errors as well and perform output result formatting accordingly.

Before we start Implementation of the services, let us think of the data we are going to send back to a user. For example, our model class “Price” has two fields “PriceId” and “ProductId” that we use for obtaining data from the database, but they mean nothing for users. More than that, we can occasionally uncover some sensitive data if our APIs respond with the entire entity. Besides that, we will use “Price” field to return a price value that is more common when working with pricelists. And we will use the “Id” field to return ProductId. “Id” name will correspond to the name of the API’s parameters for product identification (which will be observed in “Controllers” section).

So, it is a good practice to create a Data Model for output with a limited set of fields.

Let us create a new folder, ViewModels and add two classes there:

Hide   Copy Code

```
namespace SpeedUpCoreAPIExample.ViewModels
{
    public class ProductViewModel
    {
        public int Id { get; set; }
        public string Sku { get; set; }
        public string Name { get; set; }
    }
}
```

and

Hide   Copy Code

```
namespace SpeedUpCoreAPIExample.ViewModels
{
    public class PriceViewModel
    {
        public decimal Price { get; set; }
        public string Supplier { get; set; }
    }
}
```

These classes are shortened and safe versions of entity classes Product and Price without extra fields and with adjusted fields names.

Services interfaces

One service can do all the job, but we will create as many services as repositories. As usual, we start from declaring services method in interfaces.

Right click on the Interfaces folder and create a new class, IProductsService, with the following code:

Hide   Copy Code

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Interfaces
{
    public interface IProductsService
    {
        Task<IActionResult> GetAllProductsAsync();
        Task<IActionResult> GetProductAsync(int productId);
        Task<IActionResult> FindProductsAsync(string sku);
        Task<IActionResult> DeleteProductAsync(int productId);
    }
}
```

And then, IPricesService with the code:

Hide   Copy Code

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Interfaces
{
    public interface IPricesService
    {
        Task<IActionResult> GetPricesAsync(int productId);
    }
}
```

Implementation of the services

Create a new folder, Services, and add a new class, ProductsService:

Hide   Shrink   Copy Code

```
using Microsoft.AspNetCore.Mvc;
using SpeedUpCoreAPIExample.Interfaces;
using SpeedUpCoreAPIExample.Models;
using SpeedUpCoreAPIExample.ViewModels;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Services
{
    public class ProductsService : IProductsService
    {
        private readonly IProductsRepository _productsRepository;

        public ProductsService(IProductsRepository productsRepository)
        {
            _productsRepository = productsRepository;
        }

        public async Task<IActionResult> FindProductsAsync(string sku)
        {
            try
            {
                IEnumerable<Product> products = await _productsRepository.FindProductsAsync(sku);

                if (products != null)
                {
                    return new OkObjectResult(products.Select(p => new ProductViewModel()
                    {
                        Id = p.ProductId,
                        Sku = p.Sku.Trim(),
                        Name = p.Name.Trim()
                    }
                    ));
                }
                else
                {
                    return new NotFoundResult();
                }
            }
            catch
            {
                return new ConflictResult();
            }
        }

        public async Task<IActionResult> GetAllProductsAsync()
        {
            try
            {
                IEnumerable<Product> products = await _productsRepository.GetAllProductsAsync();

                if (products != null)
                {
                    return new OkObjectResult(products.Select(p => new ProductViewModel()
                    {
                        Id = p.ProductId,
                        Sku = p.Sku.Trim(),
                        Name = p.Name.Trim()
                    }
                    ));
                }
                else
                {
                    return new NotFoundResult();
                }
            }
            catch
            {
                return new ConflictResult();
            }
        }

        public async Task<IActionResult> GetProductAsync(int productId)
        {
            try
            {
                Product product = await _productsRepository.GetProductAsync(productId);

                if (product != null)
                {
                    return new OkObjectResult(new ProductViewModel()
                    {
                        Id = product.ProductId,
                        Sku = product.Sku.Trim(),
                        Name = product.Name.Trim()
                    });
                }
                else
                {
                    return new NotFoundResult();
                }
            }
            catch
            {
                return new ConflictResult();
            }
        }
        public async Task<IActionResult> DeleteProductAsync(int productId)
        {
            try
            {
                Product product = await _productsRepository.DeleteProductAsync(productId);

                if (product != null)
                {
                    return new OkObjectResult(new ProductViewModel()
                    {
                        Id = product.ProductId,
                        Sku = product.Sku.Trim(),
                        Name = product.Name.Trim()
                    });
                }
                else
                {
                    return new NotFoundResult();
                }
            }
            catch
            {
                return new ConflictResult();
            }
        }
    }
}
```

And the PricesService:

Hide Shrink Copy Code

```
using Microsoft.AspNetCore.Mvc;
using SpeedUpCoreAPIExample.Interfaces;
using SpeedUpCoreAPIExample.Models;
using SpeedUpCoreAPIExample.ViewModels;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Services
{
    public class PricesService : IPricesService
    {
        private readonly IPricesRepository _pricesRepository;

        public PricesService(IPricesRepository pricesRepository)
        {
            _pricesRepository = pricesRepository;
        }

        public async Task<IActionResult> GetPricesAsync(int productId)
        {
            try
            {
                IEnumerable<Price> pricess = await _pricesRepository.GetPricesAsync(productId);

                if (pricess != null)
                {
                    return new OkObjectResult(pricess.Select(p => new PriceViewModel()
                    {
                        Price = p.Value,
                        Supplier = p.Supplier.Trim()
                    }
                    )
                    .OrderBy(p => p.Price)
                    .ThenBy(p => p.Supplier)
                    );
                }
                else
                {
                    return new NotFoundResult();
                }
            }
            catch
            {
                return new ConflictResult();
            }
        }
    }
}
```

Data integrity between the Products and Prices tables

In the ProductsService we have the DeleteProductAsync method, which invokes an appropriate method for the PricesRepository to delete a product data row from the Products table. We have also established a relationship between the Products and Prices tables by means of the FK_Prices_Products foreign key. Since the FK_Prices_Products foreign key has a CASCADE delete rule, when deleting records from the Products table the related records from the Prices table will also be deleted automatically.

There are some other possible approaches to enforce data integrity without Foreign Keys with cascade delete in a database. For instance, we can configure Entity Framework to perform the cascade deleting with the WillCascadeOnDelete() method. But this also demands to reengineer our data model. Another approach is to realize a method DeletePricessAsync in the PricesService and call it with the DeleteProductAsync method. But we must think of doing this in a single transaction, because our application can fail when a Product has already been deleted but not the prices. So, we can lose data integrity.

In our example we use the Foreign Keys with cascade delete to enforce data integrity.

Note! Obviously, in a real application, the DeleteProductAsync method should not be invoked so easily because the important data can be lost by accident or intentionally. In our example, we use it just to expose the data integrity idea.

Services pattern

In the constructor of a service, we inject appropriate repository via dependency injection. Each method gets data from the repository inside try – catch construction and returns the IActionResult accordingly to the data processing result. When returning a dataset, the data translates to a class from the ViewModel folder.

Note, that responses OkObjectResult(), NotFoundResult(), ConflictResult() and etc. correspond to Controller's ControllerBase Ok(), NotFound(), Conflict() methods respectively. A Service sends its response to a Controller with the same IActionResult type, as a Controller sends to a user. This means, that a Controller can pass the response directly to a user without the necessity to adjust it.

The last step for the services is to declare them in the Startup class. Add using directive:

Hide Copy Code

```
using SpeedUpCoreAPIExample.Services;
```

and declare Services in ConfigureServices method after repositories declaration:

Hide Copy Code

```
services.AddTransient<IProductsService, ProductsService>();
services.AddTransient<IPricesService, PricesService>();
```

As our services are lightweight and stateless, we can use the Transient Services scope model.

The final ConfigureServices method at this stage is:

Hide Copy Code

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddDbContext<DefaultContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("DefaultDatabase")));

    services.AddScoped<IProductsRepository, ProductsRepository>();
    services.AddScoped<IPricesRepository, PricesRepository>();

    services.AddTransient<IProductsService, ProductsService>();
    services.AddTransient<IPricesService, PricesService>();
}
```

The Controllers

In our design pattern we have not left much for the controllers but just to be gateways for incoming requests - no business logic, data access, error handling and so on. A controller will receive incoming requests according to its routes, call an appropriate method of a service, that we injected via dependency injection, and returns the results of these methods.

As usual, we will create two small controllers instead of a big one, because they work with logically different data and have different routes to their APIs:

ProductsController routes:

[HttpGet] routing

- */api/products – returns the whole list of products;*
- */api/products/1 – returns one product with Id = 1;*
- */api/products/find/aaa – returns list of products which Sku field consists of parameter "sku" value "aaa".*

[HttpDelete] routing

- */api/product/1 – removes product with Id = 1 and its prices (cascading).*

PricesController routes:

[HttpGet] routing

- */api/prices/1 – returns list of prices of product with Id = 1.*

Creating controllers

Right click the Controllers folder then select Add Item > Class > ProductsController.cs and change the text for:

Hide Shrink ▲ Copy Code

```
using Microsoft.AspNetCore.Mvc;
using SpeedUpCoreAPIExample.Interfaces;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : Controller
    {
        private readonly IProductsService _productsService;

        public ProductsController(IProductsService productsService)
        {
            _productsService = productsService;
        }

        // GET /api/products
        [HttpGet]
        public async Task<IActionResult> GetAllProductsAsync()
        {
            return await _productsService.GetAllProductsAsync();
        }

        // GET /api/products/5
        [HttpGet("{id}")]
        public async Task<IActionResult> GetProductAsync(int id)
        {
            return await _productsService.GetProductAsync(id);
        }

        // GET /api/products/find
        [HttpGet("find/{sku}")]
        public async Task<IActionResult> FindProductsAsync(string sku)
        {
            return await _productsService.FindProductsAsync(sku);
        }

        // DELETE /api/products/5
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteProductAsync(int id)
        {
            return await _productsService.DeleteProductAsync(id);
        }
    }
}
```

Controller's name should have "Controller" suffix. Using the directive [Route("api/[controller]")] means that the basic route of all ProductsController, the controller's API will be /api/products

The same for the PricesController controller:

Hide   Copy Code

```
using Microsoft.AspNetCore.Mvc;
using SpeedUpCoreAPIExample.Interfaces;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Contexts
{
    [Route("api/[controller]")]
    [ApiController]
    public class PricesController : ControllerBase
    {
        private readonly IPricesService _pricesService;

        public PricesController(IPricesService pricesService)
        {
            _pricesService = pricesService;
        }

        // GET /api/prices/1
        [HttpGet("{Id}")]
        public async Task<IActionResult> GetPricesAsync(int id)
        {
            return await _pricesService.GetPricesAsync(id);
        }
    }
}
```

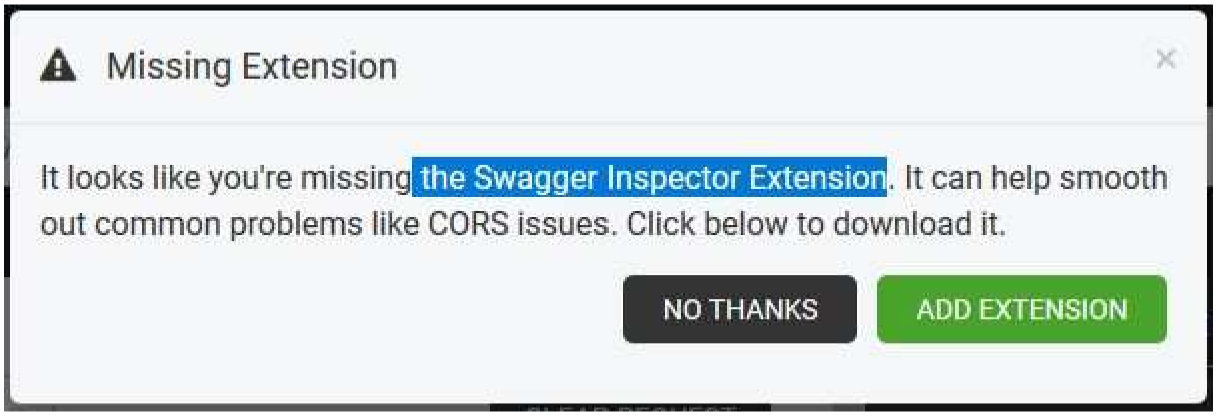
Now everything is almost ready to start our application for the first time. Before we launch the application, let us look inside the launchSettings.json file in folder /Properties of the application. We can see, "launchUrl": "api/values". Let us remove that "/values". In this file we can also change a port number in the applicationUrl parameter: "applicationUrl": "http://localhost:49858/", in our case, the port is 49858.

And we can remove the ValuesController.cs controller from the Controllers folder. The controller was created automatically by the Visual Studio and we do not use it in our application.

Start the application by clicking Main Menu > Debug > Start Without Debugging (or press Ctrl+F5). The application will be opened in the Internet Explorer browser (by default) and the URL will be http://localhost:49858/api

Examine the application

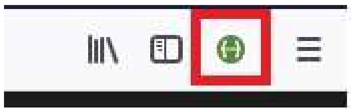
We will use the Swagger tool to examine our application. It is better to use Google Chrome or Firefox browsers for this. So, open Firefox and enter https://inspector.swagger.io/builder in the URL field. You will be asked to install the Swagger Inspector Extension.



Missing Extension

It looks like you're missing the Swagger Inspector Extension. It can help smooth out common problems like CORS issues. Click below to download it.

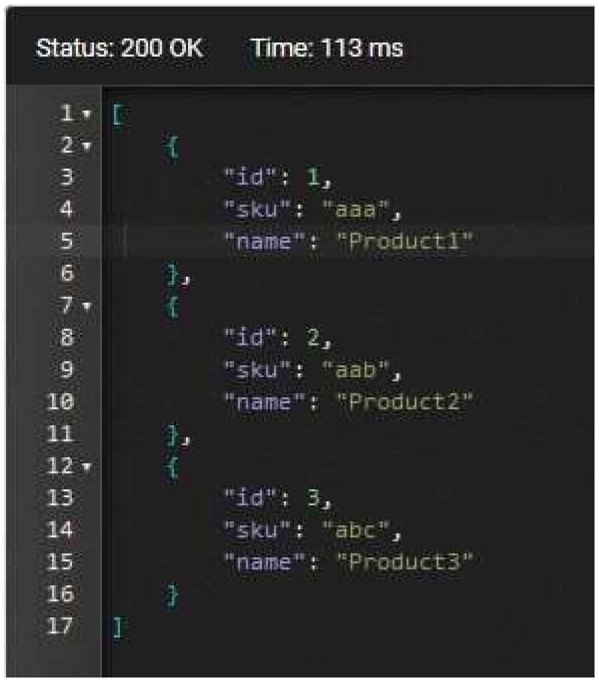
Add the Extension.



Now we have a button in the browser to start the extension. Open it, select GET method and input URL of the API

http://localhost:49858/api/products

Click Send button and you will receive a json formatted list of all products:

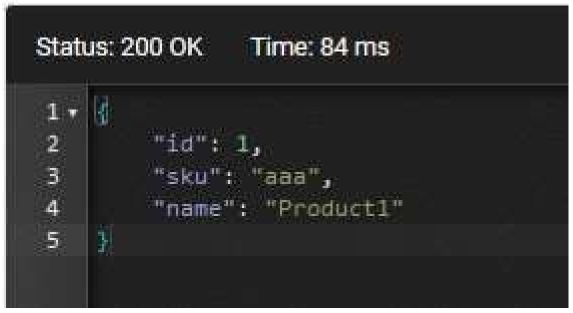


Status: 200 OK   Time: 113 ms

```
[
    {
        "id": 1,
        "sku": "aaa",
        "name": "Product1"
    },
    {
        "id": 2,
        "sku": "aab",
        "name": "Product2"
    },
    {
        "id": 3,
        "sku": "abc",
        "name": "Product3"
    }
]
```

Check the particular Product

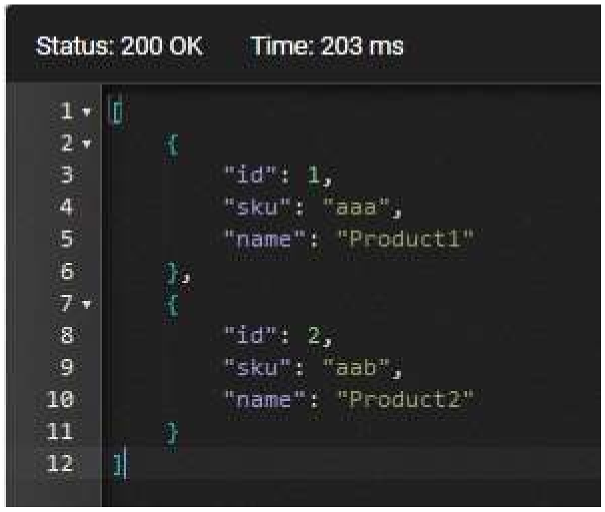
http://localhost:49858/api/products/1



```
Status: 200 OK     Time: 84 ms

{
    "id": 1,
    "sku": "aaa",
    "name": "Product1"
}
```

Find products by part of sku

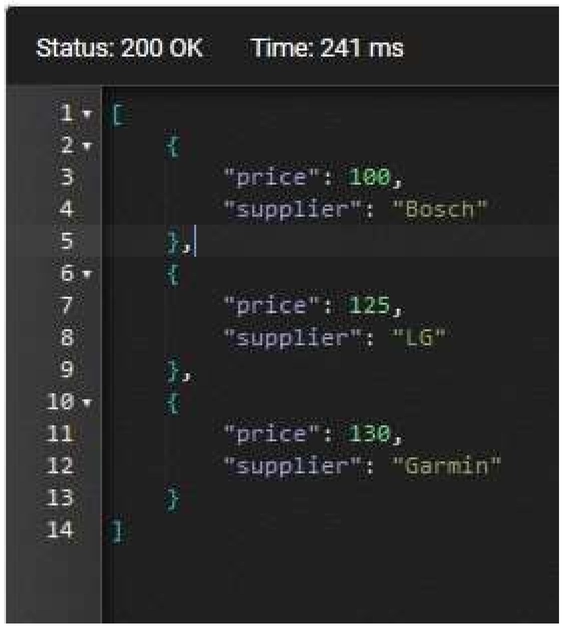
http://localhost:49858/api/products/find/aa



```
Status: 200 OK     Time: 203 ms

[
    {
        "id": 1,
        "sku": "aaa",
        "name": "Product1"
    },
    {
        "id": 2,
        "sku": "aab",
        "name": "Product2"
    }
]
```

Then check PricesController API

http://localhost:49858/api/prices/1

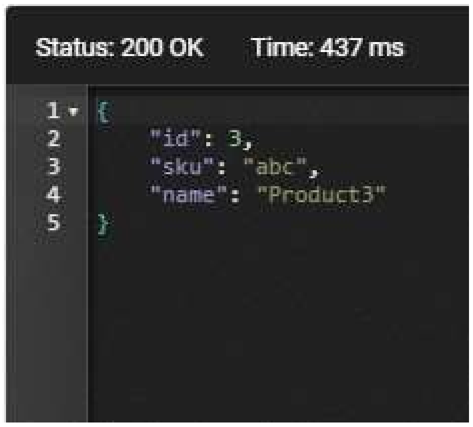


```
Status: 200 OK     Time: 241 ms

[
    {
        "price": 100,
        "supplier": "Bosch"
    },
    {
        "price": 125,
        "supplier": "LG"
    },
    {
        "price": 130,
        "supplier": "Garmin"
    }
]
```

Check Delete API

Change a method in Swagger for DELETE and call API:

http://localhost:49858/api/products/3



Status: 200 OK   Time: 437 ms

```
{
    "id": 3,
    "sku": "abc",
    "name": "Product3"
}
```

To check the deletion result we can call API http://localhost:49858/api/products/3 with GET method. The result will be 404 Not Found.

Calling http://localhost:49858/api/prices/3 will return an empty set of prices.

Summary

At last we have the working ASP.NET Core RESTful WEB API service.

So far everything has been quite trivial and just a preparation for exploring problems with the Application productivity.

But something important has already been done at this stage for increasing application performance - implementing asynchronous design pattern.

Points of interest

In Part 2 of this article we will use various approaches to increase the application's productivity.

License